SPECIFICATION

Electronic Version 1.2.8
Stylesheet Version 1.0

PROGRAMMABLE STORAGE NETWORK PROTOCOL HANDLER ARCHITECTURE

Background of the Invention

[0001] Field of the Invention

[0002] The present invention generally relates to network communications systems and, more particularly, to a programmable network protocol handler architecture that achieves high speed performance.

[0003] Background Description

[0004] Currently available network protocol handlers are implemented as hardwired logic, but there are also several programmable implementations. Custom integrated circuits (ICs), or application specific integrated circuits (ASICs), achieve good performance, but are inflexible solutions and are inefficient to handle emerging or evolving network architectures. Programmable protocol handlers are based either on a single processor implementation (which may be embedded on a chip or implemented off chip) or have multiple pico-processors controlled by a hierarchical controller (such as the IBM PowerNP network processor). All these solutions have dynamic random access memory (DRAM) external to the chip.

[0005] Programmable architecture is easier to implement than hardwired logic. Changes to specification can be easily accommodated by changing the program, and the architecture can be personalized to handle different protocols. However, current programmable engines based on single reduced instruction set computer (RISC) architecture (such as the QLogic ISP2300 Fibre Channel processor) are not fast enough for handling multi-gigabit/second (e.g., 10 Gb/s) transmission bit-rates.

[0006] In the prior art, protocol handler chips consist of hardwired logic that handles time critical operations, buffers, and a processor engine for executing high-level protocol commands and managing the overall chip function. In such chips, typically only one resource is assigned to execute a particular protocol task. Thus, such resources can become performance bottlenecks when the network traffic workload is increased.

Brief Summary of the Invention

[0007] It is therefore an object of the present invention to provide an architecture that achieves high speed performance in a network protocol handler. In the preferred embodiment, the Fibre Channel protocol is implemented; however, the architecture can be the basis for implementing many other network protocols; e.g., Giga-Ethernet, Infiniband or Internet protocol (IP) over small computer system interface (i-SCSI).

[0008] This invention introduces a programmable system based on a multiple processor and multiple threads architecture containing embedded DRAM. The architecture offers the advantages of being easily programmable, and providing balanced computing resources. This is particularly important for the future implementation of emerging protocols (such as, for example, 10 Gb/s Fibre Channel, 10 Gb/s Ethernet, etc.), since the functionality and computing requirements of such networks are still not completely specified. A single network processor may not have the computing capabilities to handle all the requirements of a very complex protocol, while bundling multiple processors to work in parallel on a single job may be hard to accomplish.

[0009] In the architecture according to the invention, performance is achieved via a combination of parallelism and pipelining, along with specialized front-end logic that handles time critical protocol operations. More specifically, a front-end hardwired logic at the network interface handles time critical operations, such as encoding/decoding, cyclic redundancy check (CRC) generation/checking, interpretation of some header bits and functions, etc. Multiple processors are used. These processors are interconnected via the processor's high-speed interconnect, which can be implemented as a ring, switch, bus, or any other processor interconnect architecture. Each processor has multiple threads, each capable of fully executing programs, and each processor's memory is globally accessible by other processors. Each processor has a memory hierarchy, consisting of embedded dynamic random access memory (DRAM) and can include data caches, instruction caches, scratch pad static random access memory (SRAM), or any combination of these memory elements. Threads within a processor are assigned the processing of various protocol functions in a parallel/pipelined fashion. Data frame processing is done by one or more of the threads to identify related frames. Related frames are dispatched to the same thread so as to minimize the overhead associated with memory accesses and general protocol processing.

[0010] The invention also addresses the problem of serialization bottlenecks arising from the use of single resources by using multiple on-chip resources and a method for monitoring resource activity and reallocating the workload to the resources that are being least used.

Brief Description of the Several Views of the Drawings

[0011] The foregoing and other objects, aspects and advantages will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

[0012] Figure 1 is block diagram showing the overall programmable architecture according to the invention;

[0013] Figure 2 is a block diagram showing the architecture and the memory hierarchy of one of the processors used in the system of Figure 1;

[0014] Figure 3 is a block diagram showing the logical memory management and thread allocation technique;

[0015] Figure 4 is a flow diagram showing the process in which an inbound data block (IBDB) address is added to the master queue;

[0016] Figure 5 is a block diagram showing the organization of the input queue as a ring buffer;

[0017] Figure 6 is a flow diagram showing the process by which the master thread moves the IBDB address of the frame from the master input queue to the local input queue of the particular thread;

[0018] Figure 7 is a flow diagram showing the process of allocating a thread to do the transfer of data from the inbound to the outbound processor;

[0019] Figure 8 is a flow diagram showing the process of monitoring and reallocating processor resources; and

[0020] Figure 9 is a flow diagram showing the process of monitoring and reallocating memory resources.

Detailed Description of the Invention

[0021] DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT OF THE INVENTION

[0022] Referring now to the drawings, and more particularly to Figure 1, there is shown the overall programmable architecture according to the invention. This architecture handles network traffic from one network protocol, performs traffic conversion from the first network protocol to a second network or bus protocol, and handles traffic of the second network or bus protocols. In our preferred embodiment, the architecture handles traffic from Fibre Channel and Infiniband network architectures. Connection to a network physical communication system is by means of receiver 101 and transmitter 102 supplied by a clock signal from a phase locked loop (PLL) 103. The receiver 101 and transmitter 102 are connected to hard-wired fiber channel (FC) port logic 104 which outputs frame data to an inbound FC interface (IF) 105 and receives frame data from an outbound FC interface (IF) 106.

[0023] As seen in Figure 1, the preferred embodiment of four processors 107, 108, 109, 110, connected by interprocessor high-speed interconnect 111, are assigned the tasks of handling network protocol, but any number of processors can be assigned to handle network protocol tasks, depending on the capabilities of the processors and of the complexity of the network protocol. Fibre Channel input traffic is handled by FC Inbound Processor 107, traffic into the Infiniband network is handled by the IB Inbound Processor 108, outbound traffic into the Fibre Channel network is handled by FC Outbound Processor 110, and traffic coming from the Infiniband network is handled by IB outbound processor 109. Within a processor 107, 108, 109, or 110, several identical threads perform various protocol sub-tasks.

[0024] Figure 2 shows in more detail the parallel architecture of the processors 107 through 110. Inter-processor communication is implemented by means of the high-speed interconnect interface 201 which provides bi-directional data flow between this processor and the other elements of the system shown in Figure 1. The processor incorporates embedded DRAM 202 which is a local memory of this processor, but which is accessible in part or as a whole by other processors globally. The processor contains also one or more instruction caches 203, and multiple identical thread units 204. The processor elements communicate via a high-speed interconnect device 205, which can be implemented as a local bus, switch, or any other implementation.

[0025] Each thread unit 204 incorporates a register file (Rfile) 206, a program counter, an arithmetic logic unit (ALU) 207, and logic for instruction fetching, decoding and dispatching 208. A single thread unit 204 can or can not have logic for instruction pre-fetching and storing, branch prediction mechanisms and logic for out-of-order execution and speculation. The processor can have multiple data caches (D-caches) 209, dedicating one cache block to each thread unit, or sharing a single D-cache block among several thread units 204. Instead of having a data cache, or in addition to it, an implementation can have a scratch pad memory, again shared between multiple thread units, or being local to only one thread unit 204.

[0026] Referring back to Figure 1, inbound frames are first stored in frame first-in, first-out registers (FIFOs) at the FC interface 105. These can be dual FIFOs that operate in ping-pong fashion to prevent frame overruns or can be implemented as a single FIFO. In the preferred embodiment, associated with the FIFOs are two addresses, one pointing to a memory area in the FC inbound processor 107 where the header of an incoming frame will be written into, and the other pointing to a memory area in the IB inbound processor 108, for writing the payload of the frame (i.e., the data). Or in another embodiment, only one address is associated with the FIFO block, storing both the header and the payload of the frame. These addresses are set by the FC inbound processor 107 that manages the respective memory areas. Once a frame begins arriving into the FIFO, the FIFO logic moves first the header and then the payload into the preassigned inbound data block (IBDB) areas over the high-speed interconnection device.

[0027]

As shown in Figure 3, the beginning addresses of the two parts of the IBDB block or a single address of the IBDB block is added to a master input queue 301 by the thread 303 which manages the IBDB memory area. Then, this "master" thread assigns the incoming frames to one of the fixed number of threads 304 through 305 performing FC protocol. The frame dispatching is performed by a workload allocation function.

[0028] Figure 3 shows the logical assignment of thread units of a processor 107 to protocol tasks and the logical organization of the memory 300. Memory 300 contains an area dedicated for packet storage 308, where the payload and headers of the frames are stored, an area for storing control and status blocks (SSCB and ESCB) 309, where various protocol specific information and the current status of the network traffic is stored, and area for working queues, where master input queue 301 and local input queues 306 to 307 are stored, as well as any other information required.

[0029] Several identical processor threads perform FC protocol tasks on the incoming frames. The protocol thread fetches the IBDB address of the frame from its corresponding local input queue, and uses it to access the frame header. Then, it fetches the control blocks associated with the frame from the memory (ESCB and SSCB), to begin frame processing. The FC protocol tasks include context switch, frame validation, frame reordering and acknowledgment generation. Once all required FC protocol tasks have been performed for a single frame, the IBDB address of the next frame is fetched from the local queue. If the local input queue is empty, the protocol thread goes into the "idle" state. In the "idle" state, the processor can check for new data by polling its local queue, or can wait for an interrupt or some particular external signal to get the new IBDB address.

[0030] Input processing unit

[0031] The task of the input processing unit is to copy data from the inbound FIFO to the packet memory 308, and to add the new IBDB address to the master input queue 301.This process is shown in Figure 4. The process begins by retrieving the IBDB address in function block 401. The contents of the FIFO are copied into the IBDB in function block 402. The IBDB address is then written into the master queue in function block 403 before the process loops back to function block 401.

[0032]

In one embodiment, the input processing unit can be implemented as a software application running on a dedicated thread. In another embodiment, this task can be performed by a direct memory access (DMA) unit with incorporated functionality 403 to add the IBDB address to the master input queue 301.

[0033] For each new packet, an address of the next free IBDB memory block has to be provided. This is done by reading the "first free IBDB pointer" from the chain of free IBDBs, and then adjusting the first free IBDB pointer to the next IBDB block in the chain. In the preferred embodiment, memory blocks for data and control areas are organized as a chain of free blocks, where the first element of the block contains the address of the next free memory block, but the memory can be organized in any other way.

[0034] All input queues in this architecture are organized as a ring buffers, with the first and last valid buffer locations recorded in the head and tail pointers 501 and 502 of the particular queue, respectively, as shown in Figure 5. The head and tail pointers for each queue can be implemented as dedicated registers in hardware, or can be stored in the memory on the previously determined locations. In the latter case, the addresses of the memory locations containing these pointers are determined before the processing has begun and are not changed during the processing. In addition, the queues themselves can be implemented as dedicated hardware or can be stored in the memory on the previously determined locations.

09682688.102501

[0035] Master thread

[0036] The master thread dispatches the incoming frames to one of the threads performing FC protocol. To accomplish this, some packet assignment method is used. This is accomplished by using some workload allocation function, which can be implemented by using some well known method, e.g., table lookup, round-robin, first-come, first-server, etc., or can be implemented to use data from the frame header or from the frame payload for allocating the frame to a particular thread.

[0037] To allocate the incoming frame to one of the threads performing the FC protocol, the master thread first fetches the address of the IBDB memory block of the incoming frame from the master input queue. Then, the workload allocation function is performed. If the workload allocation function requires data from the frame, these data are first fetched, and then the workload allocation function is performed. The result of the allocation function is the dispatching information, and the frame is assigned for processing to the corresponding protocol thread.

[0038] Once a frame has been assigned to a protocol thread, the master thread moves the IBDB address of the frame from the master input queue to the local input queue of the particular thread, as shown in Figure 6. The process begins by reading the master input queue in function block 601. If frame data are required for the workload allocation function, these are fetched in function block 602. A workload allocation function is performed in function block 603, and then the frame is assigned to a thread in function block 604. The IBDB is written to the local input queue in function block 605 before the process loops back to function block 601.

[0039] Protocol thread

[0040] There is a fixed number of protocol threads in our preferred embodiment. The protocol threads are identical and perform FC protocol-specific tasks. These tasks include context switch, frame validation, frame reordering and acknowledgment generation.

[0041] Referring now to Figure 7, the address of the IBDB block of a data frame is fetched from the local input queue of the protocol thread in function block 701. If a protocol thread is idle, the tail of the local queue is polled. The polling frequency is kept low, to reduce bus traffic.

[0042] To fetch the IBDB data from the local queue, the IBDB address from the location addressed by the ring buffer's tail 502 is read (see Figure 5). The ring buffer tail address 502 is adjusted to the next location, and it is checked to see if the input queue is empty by comparing the addresses of the ring buffer's head 501 and tail 502.

[0043] The data frame is accessed using the IBDB address, and protocol tasks can be performed. The protocol thread fetches several data words from the frame header in the IBDB memory block, and compares it with the cached data in decision block 702. If the comparison shows that the frame belongs to the same sequence as the previous frame processed by that thread, neither context switching nor new data fetching needs to take place. Otherwise, context switching is performed in function block 703 and required data are fetched from the corresponding exchange and sequence status control blocks (ESCB and SSCB) from memory. Checking for the context switch reduces the number of memory accesses and bus traffic significantly, thus boosting overall performance.

[0044] The frame is then checked for validity in function block 704 and a determination is made in decision block 705 as to whether the frame is valid. If it is not, the sequence is discarded in function block 706, and the process loops back to function block 701. If the frame is valid and for certain classes of service, a determination is made in decision block 707 as to whether it has been received in order. If the frame has been received out of order and it is of class 2 service, it is placed in a reorder table in function block 708, so that the sequence can be delivered in its proper order.

[0045] The frame is chained to the previously received frame from the same sequence in function block 709. The chaining and reordering mechanism enables the system to link multiple data blocks in a single data stream, keeping the information needed to find the next data block in the sequence locally in the data block.

[0046] If the frame requires a response to be generated to the sender as determined in decision block 710, such as acknowledgment or busy message, a response frame is constructed in function block 711. All required data for the response are collected in an OBDB (outbound data block) block and the pointer of the OBDB block is sent to the outbound processor 110 (Figure 1), which composes and transfers the response frame. The FC outbound processor 110 composes the response frame and transfers it to the outbound FIFO in the FC interface 106 in Figure 1. The IBDB address of the data frame is then placed in the work queue of the thread for transferring the data to the host interface FIFO 112 (Figure 1), as shown in the function block 712. This task is performed by the Infiniband inbound processor 108 (Figure 1). From here, the data are sent to the Infiniband network.

[0047] It is also an object of the present invention to allocate the protocol handler resources, i.e., processors, memories and high-speed interconnect, based on the demand requirements of the workload. The monitoring of the activity of the various resources takes place as follows.

[0048] Processor resources:

[0049]

In the FC inbound processor 107 (Figure 1) there is a master thread that assigns the processing of incoming frame headers to protocol threads. This is done by the master thread placing requests in queues stored in memory. When these queues become large, indicating that the processing of the frames has slowed down, the master thread can create request service queues in another processor's memory (such as the IB inbound processor 108), thus having threads in that processor service new incoming frames. This will reduce the workload in the FC inbound processor, thus increasing protocol handler performance.

[0050] The process is shown in Figure 8 and begins when a frame is received at the interface in function block 801. When this occurs, the inbound processor is notified and, in response, the inbound processor checks the size of its work queue in function block 802. A determination is made in decision block 803 as to whether the work queue is above a predetermined threshold. If not, the pointer to the inbound frame is placed on the work queue in function block 804, and the process loops back to function block 801 to await the next frame. If, however, the threshold is exceeded, th inbound processor checks the size of another processor's work queue in function block 805. A determination is made in decision block 806 as to whether the work queue of the other processor is above the threshold. If not, a pointer to the inbound frame is placed on the work queue of the other processor in function block 807, and the process loops back to function block 801 to await the next frame. If, however, the threshold is exceeded in the other processor's work queue, a determination is made in decision block 808 as to whether there are any other processors available. If so, the process loops back to function block 805 to check that processor's work queue; otherwise, the processor indicates to the interface to stop flow of incoming frames via pacing in function block 809. The process then loops back to function block 802 where the inbound processor again checks the size of its work queue.

[0051] Memory resources:

[0052]

Incoming frames are separated into header and payload parts, and the header is stored in the memory of the FC inbound processor 107 while the payload is stored in the memory of the IB inbound processor 108, so that it can be properly formatted for delivery to a destination in the Infiniband network. Associated with each processor's memory is a memory free list that provides pointers to available memory blocks. If the free list associated with a processor's memory is about to become empty, indicating that there is not any more available memory for storing incoming frames in that processor, the master thread that assigns memory blocks to the incoming frames can examine the free lists in other processors to determine the availability of corresponding memory. Then, the master thread can redirect the incoming traffic to a process whose memory is under utilized.

[0053] The process is shown in Figure 9 and begins with a frame being received at the interface in function block 901. When this occurs, the inbound processor is notified and, in response, the inbound processor checks the size of its free memory list in function block 902. A determination is made in decision block 903 as to whether the free memory list is empty. If not, a pointer to free memory space is sent to the interface logic and the frame is transferred into memory from the interface in function block 904, and the process loops back to function block 901 to await the next frame. If, however, the free memory list is empty, the inbound processor checks another processor's free memory list in function block 905. A determination is made in decision block 906 as to whether the free memory list of the other processor is empty. If not, a pointer to free memory space is sent to the interface logic and the frame is transferred into memory from the interface logic in function block 907, and the process loops back to function block 901 to await the next frame. If, however, the free memory list of the other processor is empty, a determination is made in decision block 908 as to whether there are any other processors available. If so, the process loops back to function block 905 to check that processor's free memory list; otherwise, the processor indicates to the interface to stop flow of incoming frames via pacing in function block 909. The process then loops back to function block 902 where the inbound processor again checks the size of its free memory list.

[0054] Interconnection resources:

[0055]

As previously stated, the high-speed interconnect among processors can be implemented as a ring, bus, or any other processor interconnect architecture, but in the preferred embodiment, the processor interconnect is implemented as dual counter-rotating rings. If a packet needs to be sent to another processor, the ring interface logic at the processor selects the ring that provides the shortest distance to the destination and places the packet into that ring. If the ring is busy at the time because another packet is in transit, it waits until the ring becomes available again. Alternatively, it can place the packet on the other ring. The performance of the interconnection rings can be optimized by balancing their workloads. This can be done by the ring interface logic which monitors the traffic on both rings by counting the packets that pass by, thus determining which ring has less traffic. The ring interface logic can then try first to place its packet on the ring that has the least traffic. This will minimize the possibility of waiting for other packets upstream and downstream.

[0056] While the invention has been described in terms of a single preferred embodiment, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.